

Software Product Lines: a realistic path to software development industrialization?

Bruno Cal¹, Henrique O'Neill²

1) Instituto Universitário de Lisboa (ISCTE-IUL), ADETTI-IUL

bruno_miguel_cal@iscte.pt

1) Instituto Universitário de Lisboa (ISCTE-IUL), ADETTI-IUL

henrique.oneill@iscte.pt

Abstract

Since the early days of Software Engineering, researchers and practitioners have sought to improve the software development process. The lack of reuse or the low productivity rates are recurring concerns. The research community and industry members have been studying the Software Product Lines paradigm as a mean to industrialize the software development process. This paper takes that premise, materializing the paradigm with applicable technologies and analyses its true potential.

Keywords: Software Product Lines, Industrialization, Component-based Development, Object-Oriented Frameworks, Model-Driven Development, Generators

1. Introduction

The Standish Group presents annually the Chaos Report with IT industry findings (Eveleens & Verhoef, 2010). The report main goal is to assess the worldwide success rate of applications development projects.

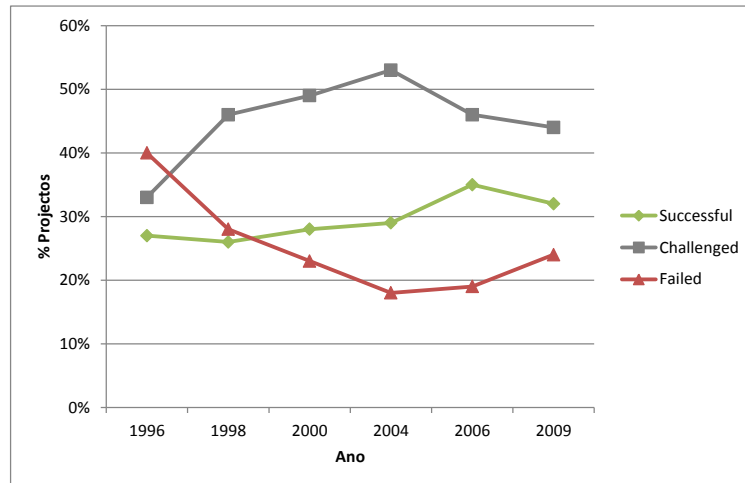


Figure 1 – Latest Chaos Report results. Adapted from (Eveleens & Verhoef, 2010)

The report sets three kinds of project classifications:

- Successful: to classify projects that end on schedule, within budget and with all the functionality initially agreed.
- Challenged: to categorize projects that failed on one of the vectors: schedule, budget or scope.
- Failed: to classify projects that are not accepted by the customer or do not end at all;

As we can see by looking at Figure 1 the results are not very exciting. On one hand, the “Successful” projects went from 26% in 1996 to 32% in 2009, beyond the obvious low global value, there’s only a 6% increase in 13 years. On the other hand, in 2009 we held a high rate (44%) of projects that fail at some assessment criteria – schedule, budget or scope – and still kept a 24% of miscarried projects.

Despite some authors criticism [e.g. (Rise & Figures, 1994)] concerning the methods and metrics used by the Standish Group to elaborate the report, there’s a widespread perception that even nowadays it is extremely difficult to finish a project that wins in all the three cornerstones.

2. Industrialization

Countless researchers have proposed several explanations for the previously stated results. The “fast changing” environment that usually comprises the target business or the current information systems complexity are two of the most outlined reasons to explain this issue.

Many of these researchers have pointed to an inevitable evolution towards the industrialization of software development. But what does software industrialization mean? As Jack Greenfield *et al* narrated in (Greenfield, Short, Cook, & Kent, 2004):

“We cannot know with certainty until it [industrialization] happens, of course. We can, however, make educated guesses by looking at how this industry has evolved to date. We can also gain some insight by looking at what industrialization has meant in other industries and by comparing them with this industry, to reason about how our experience may be similar to or different from theirs”.

Trying to materialize the route traced by Greenfield, to capture the meaning of software development industrialization, we need to analyse the traditional development approach and to examine the industrialization process followed by other industries. Such approach led us to some conclusions.

Concerning to the traditional development approach, there are two recurring criticisms [e.g. (Lenz & Wienands, 2006) or (Kleppe, Warmer, & Bast, 2003)]:

- The lack of reuse. Indeed, in the traditional software development approach we do not reuse, or at least, we do not take reuse as a conscious, programmed and appellant practice. This line of attack ends repeatedly in constant reinvention where variants of the same problem are addressed differently.
- The usage of low abstraction techniques. Many times, our business analyses are supported in UML models (like class or sequence diagrams) or entity relationship diagrams. These modelling techniques have too much technical inspiration - using concepts like inheritance or generalization - that have nothing to do with the problem domain. These semantic gaps ends, frequently, in mismatches between the problem characteristics and the features offered by the solution.

Now looking for the industrialization process in others industries we can identify three milestones that have a direct representation in the software universe (Czarnecki & Eisenecker, 2000).

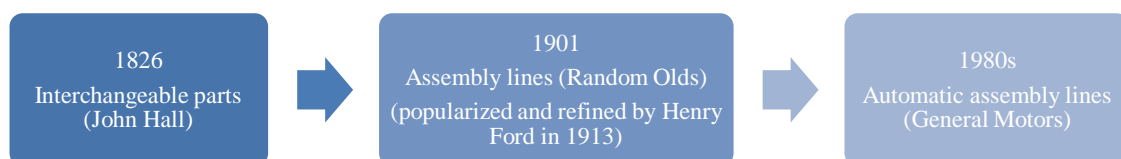


Figure 2 – Major industrialization milestones. From (Czarnecki & Eisenecker, 2000).

The innovations identified on the milestones in Figure 2 have already a representation in the software development world.

Interchangeable parts are a fundamental piece of any industrialized process because they represent the ability to create reusable components that can be included in various products allowing significant improvements on the necessary effort implied on each of them. This “component concept” exists in software engineering since its early days. McIlroy 1968 paper “Mass product software components” (McIlroy, 1968) is probably the first manifestation of this circumstance.

Based on this component notion, a development approach was born, many times referred to as component-based development. This approach is characterized by Sommerville (Sommerville, 2006) as the process of defining, implementing and integrating or composing loosely coupled independent components into systems.

Most of the existing variants of this development approach have incorporated an idea of assembly line – the second industrialization milestone. Each component has a well-defined interface and very strict configuration/adaptations instructions. Thus, the application development process, i.e. the components personalization and integration, ought to follow a pre-established plan. Guided by this plan - production plan - the application developers produce the application such like any other factory worker executes pre-defined steps to operate machinery, tools and components in order to build the desired product.

The third industrialization milestone is the assembly line automation. As stated, the assembly line is a concept that has an analogous concept in the software development, which applies to the automation of this process. Nowadays, many software development tools like Integrated Development Environments (IDEs) have many versatile extension mechanisms that allow automating various steps in the development/integration process. It’s a little bit utopic to imagine a complete automated assembly line of a complex application (as any complex assembly line in others industries) but it’s very realistic to automate a relevant portion of any production plan obtaining significant productivity improvements from this possibility.

Therefore, after this brief explanation, a possible and natural assumption is: if, by now, we’ve discovered the necessary practices, the software development process is already industrialized. In fact, this is not necessarily true. It’s correct that we already have methods and techniques that enable, or try to enable, industrialization but the day-to-day development practice (and reports like the Chaos Report) proves that reality is fairly distant from any industrialized scenario.

But what reasons may justify the disparity between the theoretical scenario and reality? It is true that component-based development never had a massive adoption from the software industry. (Pfleeger & Atlee, 2006) summarized some technical problems that are recurrently associated with this kind of development approach and, somehow, justify its lack of popularity:

- Generally it takes extra time to make a component general enough to be reusable;
- It’s difficult to create component repositories with efficient search mechanisms (sometimes it’s faster to build a small component then search for an existing one);
- It’s difficult to document the degree of quality assurance and testing that have been done to the component;
- It’s not clear who is responsible if a reused component fails or needs to be updated;
- It can be costly and time-consuming to understand and reuse a component written by someone else;
- There is often a conflict between generality and specificity.

Despite the technical reasons that set up undeniably barriers to any component-based development, there’s also a lack of process support. A component-based approach requires an appropriate development process that, in several cases, is disregarded.

These component-based approaches glitches, as well as the software industrialized vision, motivated the (re)appearance of an alternative development paradigm, the Software Product Lines.

3. Software Product Lines

Software product lines are emerging as a viable and important development paradigm allowing companies to realize order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers (Software Engineering Institute, 2010).

Over the past two decades, this approach has been largely applied with great success by worldwide companies (e.g. Philips, Boeing or Nokia) to the embedded applications development. In the embedded field, the figures are encouraging, encompassing 10x productivity and quality increases and 60% cuts on costs (Pohl, Böckle, & Linden, 2005).

The following Product Line definition is adopted in the paper: *A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* (Clements & Northrop, 2001).

In other words, a software product line is a reuse-centric development approach focused on developing the right infrastructure and assets enabling to produce a set of applications that share common features (applications in the same product line).

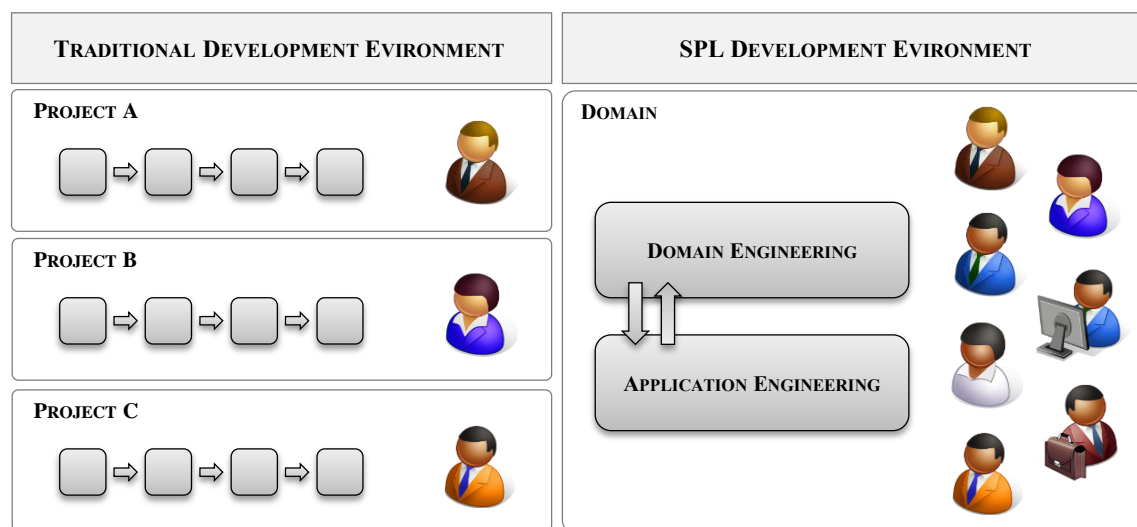


Figure 3 – Traditional vs SPL development environment

As presented in Figure 3, the traditional environment assigns a project to every customer, eventually in different domains with an entirely effort replication. This diversity has a negative impact on the amount of potential reuse. The same figure (right side) shows a conceptual environment to a product line oriented development.

To support the desired level of reuse, the SPL is focused on a specific domain. A domain is a very elastic concept; it can represent a business area, a set of related problems, a collection of applications or an area of knowledge with common terminology (Harsu, 2002). An important characteristic that is mandatory to all referred “domain forms” is the ability to provide the means to set a concrete domain boundary definition that allows setting the scope of the product line.

As Figure 33 demonstrates, an SPL environment includes two interconnected processes: Domain Engineering and Application Engineering.

3.1. Domain Engineering

The Domain Engineering (DE) is the core process of an SPL project. It has the responsibility to ensure that a proper product line is conceived and that the means to operationalize its development are available. Typically, this intent includes performing the following macro activities:

- Domain Scoping and Analysis: The domain engineering process starts out by taking a detailed analysis of the target domain. Unlike the traditional approach that focus the attention in a particular customer business, in SPL, this analysis is carried out focused strictly on the domain, totally abstracted from any concrete business peculiarities defined by customers' existing in that domain.
- Product Line Definition: this is one of the most important activities in any SPL program. It handles the variability management by setting which features should be included in all members in the family and what features, and in what circumstances, must be incorporated as variable features. This characterization is known as the product line scope.
- Building infrastructure and reusable assets: the features (common and variable) shared by the family members must be implemented through a reusable platform (common features) and a set of reusable components (variable features). To ensure the correct application production based on these assets, an appropriate infrastructure and production plan should be developed and delivered as well.

3.2. Application Engineering

Application Engineering is the process that is in charge of instantiating the product line, which means, it will build the member of the application family using the infrastructure, methods and assets made available by the Domain Engineering process.

Naturally, these two processes do not make sense alone. The Application Engineering team needs the infrastructure production and reusable assets to develop applications and the Domain Engineering team needs feedback about the product lines features or the infrastructure suitability.

4. SPL Supporting Technologies

An SPL is a development paradigm, agnostic to any technology that can be applied to its implementation. However, many SPL proposals [e.g. Software Factories – more information in (Greenfield, Short, Cook, & Kent, 2004)] are based in some existing techniques and technologies like:

- Object-oriented frameworks;
- Model-driven development;
- Automation mechanisms (e.g. Generators).

4.1. Object-oriented frameworks

In many SPL projects, the common structure between members of the family is implemented using an object-oriented framework. As defined by Fayad (Fayad, 1997), *a framework is a reusable, “semi-complete” application that can be specialized to produce custom applications.*

The common features plus the pre-established variability between applications in the same family is reflected in the framework. The applications will be made by the framework reusable structure plus the instantiation of such variability. There are two main approaches to achieve this (Johnson, 1991):

- White-box frameworks: the application specific behaviour is set through extension/implementation of specific abstract classes/interfaces in the framework;
- Black-box frameworks: the application is based, exclusively, on the composition of pre-existing components provided by the framework.

These options are not mutually exclusive. They can and, in many cases, they are used together because both have advantages and disadvantages. The white-box approach is more flexible but requires the application developer to have access and to understand the framework (complex) code; instead, a black-box framework is easier to instantiate but requires an enormous domain knowledge and implementation effort to allow the early deployment of all necessary assets to build an application.

An alternative approach to object-oriented framework that is gaining notoriety in the industry is to build family platforms in a service-oriented manner. This way, the family members are built mostly by composition of services that the product line platform provides. Several reasons may justify this alternative approach, but these two are particularly relevant:

- The most common implementation of service oriented platforms is based on web services. Taking into account that this technology uses “almost universally” accepted XML based standards (like SOAP¹, WSDL² or UDDI³), the lack of standards to describe components and its interactions, is automatically overcome;
- With respect to enterprise information systems, services, when compared to components, have the additional advantage to be closer to the domain semantics. In many cases, platform services have a direct mapping to a specific business process/service.

4.2. Model-driven development

Model-Driven Development (MDD) aims to leverage models to generate the specified software system (Demir, 2006). In other words, MDD is a technique that sets models as the core of the application development.

We already tend to use models extensively in most project lifecycle. Though, the way we do this has some disadvantages (Kleppe, Warmer, & Bast, 2003):

- We use different models to express different views of the same (part of a) system, so models have a strict relation between them. The consequence is that changes in one model may obligate (manual) changes of others. This synchronization is very hard to maintain, especially, in bigger and complex projects;
- Modeling is restricted to a descriptive perspective. Traditionally we do not take models seriously. We use them as a mean to describe systems aspects, like structure or dynamic behavior, but forget about the potential behind automatic interpretation of well-formed models and artifact generation based on that information.

¹ Simple Object Access Protocol

² Web Services Description Language

³ Universal Description, Discovery and Integration

Additionally, the languages that we use to create our models (e.g. UML) are highly bounded with implementation issues and programming languages. This has a huge impact on the level of abstraction that guides the traditional development process causing a distorted problem analysis due to the technical influences that the modeling language embodies.

In a domain oriented approach, like SPL, one of the most used methods to materialize MDD is through Domain-Specific Modeling Languages (DSML). The main distinction between these languages and General Purpose Modelling Languages (like UML) is concerned with the highly expression capability that they provide in a specific (small) domain as opposed to a general applicability to a wide variety of areas. The DSML increased specificity can raise the level of abstraction of models that are being made mostly with domain concepts.

A DSML consists of an abstract syntax, concrete syntax and semantics (Lenz & Wienands, 2006). The abstract syntax defines the elements, relationships between and rules that the language provides.

Abstract syntax characterizes, in an abstract form, the kinds of elements that make up the language and the rules to combine those elements (Greenfield, Short, Cook, & Kent, 2004). In modeling languages, this is usually achieved by meta-modeling. A meta-model is a model that formalizes the abstract syntax of the languages, i.e. it is a model containing all the elements, relationships and rules that the language has to offer.

The concrete syntax establishes the way we can interact with the language, mapping the abstract elements – set out by the abstract syntax – to any visual representation (e.g. geometry shape, picture or text). The semantics embodies the meaning of the language elements which, in a domain-specific language, it's not so difficult to acquire because the language elements derive directly from the domain and consequently share the same meaning with it.

4.3. Generators

We use generators daily on our development tasks. We use them when drag-and-drop a control to a form, generate code from any UML model or even when we compile source code in order to produce a binary file.



Figure 4 – Generation model

As presented in Figure 4, a generator is a very simple concept. From a high level and simplistic perspective, a generator is a special application that takes an input and performs a set of transformations in order to create a desired output. These transformations may (Czarnecki & Eisenecker, 2000):

- Add (refinement) or remove (abstraction) detail from the input (Vertical Transformation);
- Alter the structure of the input but do not change the level of abstraction (Horizontal Transformation);
- Perform the two operations simultaneously (Oblique Transformation).

Generators can act isolated from any model driven environment but combined with model-driven approach generators can bridge the wide gap between the high-level, intentional system description and the executable (Czarnecki & Eisenecker, 2000).

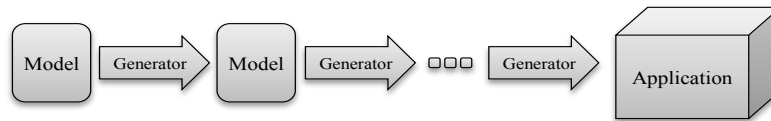


Figure 5 – Models and generators in a MDD environment

A model formalizes concepts in a well-defined, structured, way. Generators can be fed with these well-defined models and perform operations to transform them in other well-defined models. This process can be recursive till the ultimate generation that sets the application source code. This is the scenario represented in Figure 5.

One of the most well-known MDD approaches that are based on recursive model transformations is the Model-Driven Architecture (MDA). MDA was proposed by the Object Management Group as a model-driven development process based on standards developed by the institute (e.g. UML).

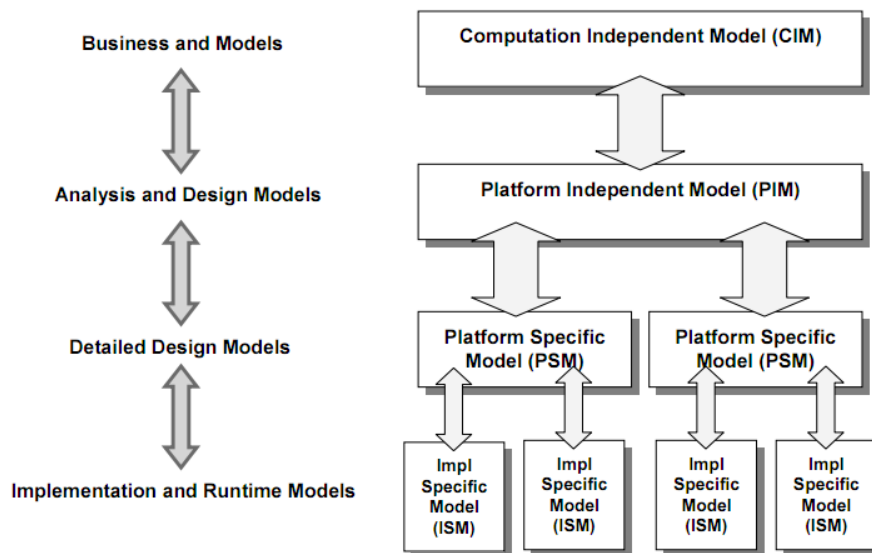


Figure 6 – Layers and Transformation of MDA. From (Brown, Conallen, & Tropeano, 2005)

Figure 6 represents the types of models that MDA considers – CIM, PIM, PSM and ISM – and the correspondent abstraction layer – Business, Analysis and Design, Detailed Design and Implementation and Runtime. The CIM model is manually developed and represents a high level problem description constituted exclusively by domain concepts. The remaining models, located on lower abstraction levels, are produced by generators (represented by the models connecting arrows) configured with appropriate transformation rules. Needless to say, these rules are extremely difficult to define. More information about MDA can be found on (Kleppe, Warmer, & Bast, 2003) or (MELLOR, Scott, Uhl, & Weise, 2004).

Different approaches combine generators with architectural frameworks and domain-specific modelling languages.

Usually, frameworks require a significant development effort due to the extensive domain knowledge that is required to anticipate variations and the wide-range of technical issues that need to be addressed. But the noteworthy effort is not restricted to the development framework.

Learning how to correctly use a non-trivial framework is a difficult and time-consuming activity (Santos, Koskimies, & Lopes, 2008). This is where models and generators can help.

So, a framework embodies features variation. According to the adopted variability mechanisms, in order to produce an application, the application developer needs to “consume” the framework by extending, defining or composing features provided by it. Typically, this error-prone activity is done mainly by coding the extensions or using available configuration mechanisms.

Alternatively, a model-driven framework consuming, turns this coding/configuration activity in a modelling activity.

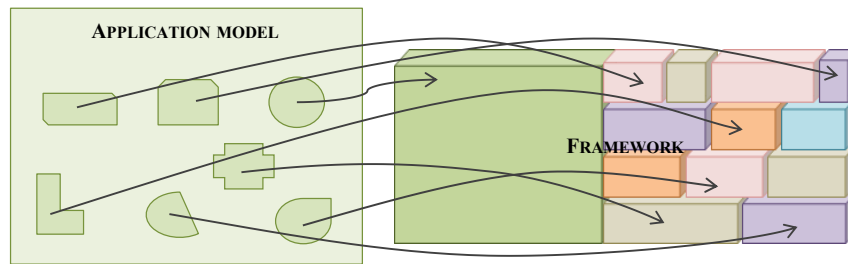


Figure 7 – Application model and product line framework

Figure 7 represents an example view of a model-driven framework instantiation. The represented “Application Model” has elements that are mapped with the features provided by the product line framework. These modelling elements (concepts) are specific to this context, i.e. to this domain framework, and, naturally, belong to a domain-specific modelling language. Therefore, our model constitutes a framework instantiation description. Generators are then applied to materialize this high level description. This means that generators are responsible for producing the necessary code to complete and configure the framework according to the elements specified in the application model.

There are various advantages associated with this approach (Kelly & Tolvanen, 2007):

- The modelling concepts correspond to business concepts so generally no further training is needed;
- The modelling language embodies business rules so the application modelling is dynamically checked accordingly;
- Generators can produce high quality code, which is a key difference from generic generators. In domain specific generators the developer can include the organization best practices in the generated code because he has absolute control over the generator;
- Generators are versatile tools. Application models can be used to produce applications but also to create other supporting development assets, like documentation, test cases or deployment packages;
- As said before, manual frameworks instantiations are complex and time-consuming. Even if we consider the time and effort spent on developing the modelling language and generator, frameworks instantiations based on models are quicker, simpler and drastically reduce errors.

5. SPL Development Methodology - SPLUP

Our research have been focusing on creating a methodology to integrate the software product line development paradigm with existing technologies, particularly, architectural frameworks (e.g. object-oriented frameworks), model-driven development and automation techniques. The

Software Product Line Unified Process (SPLUP) is the main outcome of this research orientation.

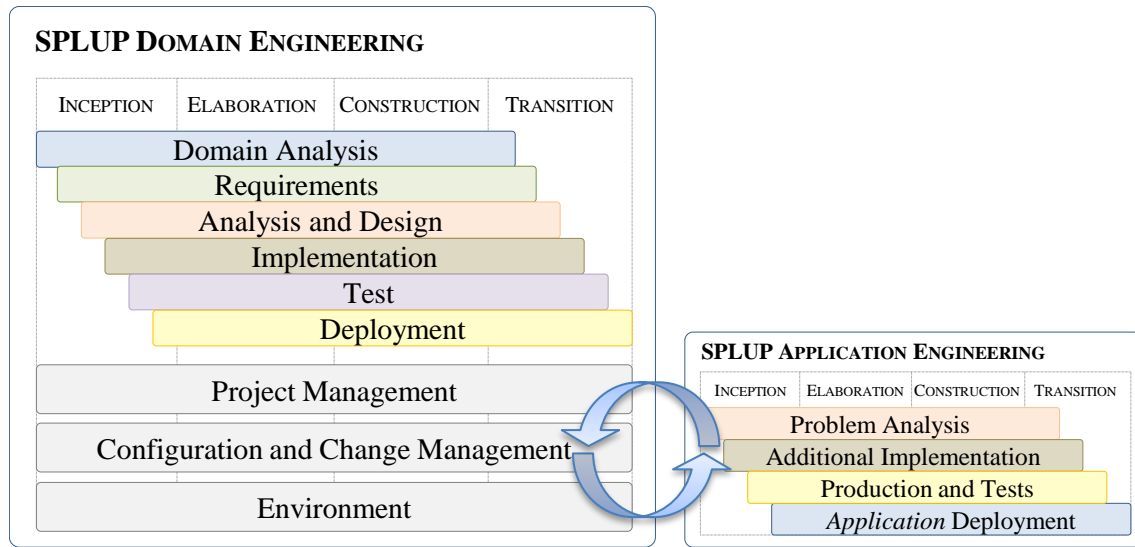


Figure 8 – SPLUP development model

As the name, and process model presented in Figure 8, suggests, SPLUP has a substantial inspiration in methodologies based on the Unified Process, namely RUP and OpenUP. This option is based, essentially, in two order of reasons: to ease the methodology acceptance by any organization that is used to traditional approaches and to ensure a solid and highly tested process background to our proposal.

Such as any SPL program, SPLUP has two macro processes: SPLUP Domain Engineering and SPLUP Application Engineering. The two processes are divided into 4 phases – Inception, Elaboration, Implementation and Construction – with associated milestones that should be assured to consider the respective phase completed.

Like Figure 8 demonstrates the process has disciplines that should be carried out along the project lifecycle. Much of our work has been centred on the definition of these disciplines and the associated workflows, artifacts and roles.

Besides the methodological perspective that few SPL proposals address, SPLUP provides a process based model to develop product line oriented applications set on frameworks, DSMLs and generators. This has the following impact over the two main processes:

- SPLUP Domain Engineering – the desired variability of the product line features should be reflected through an architectural framework (core features) and a set of framework specific components (variable features). Then, the static and dynamic perspective of this domain-specific elements should be reflected in a domain specific modelling language(s) through the corresponding meta-model(s). Formerly, development tools should be adapted and generators must be developed to transform these domain specific models into the framework and component instantiations (code). SPLUP considers the impact of this implementation architecture from early activities like requirements definition or analysis and design;
- SPLUP Application engineering – The application team interacts with the customer. They seek a solution that, at the same time, suits the customer needs and the features provided by the product line. In cases where the agreed solution features are entirely available from the product line, the application production is done exclusively by modelling the application through the DSMLs and adapted development tools

(including generators) that the engineering team provided. In cases where the desired features are not provided by the product line, the application team has to move forward with additional development. Then, two scenarios may occur:

- The new developed assets are included in the product line and framework, DSMLs and tools have to be updated accordingly. In this case, the application production still remains exclusively a modelling activity;
- The new developed assets are so context specific that they are not incorporated into the product line. In this kind of scenarios, the application developer has to generate part of the application through modelling and assemble the newly components manually.

These two core processes are described with higher detail in the next sections of this work.

5.1. SPLUP Domain Engineering

The Domain Engineering is the core of the SPLUP. Its main purpose consists in delivering all the necessary assets to the application engineering team to develop applications from the product line.

We decided to support the process into the work structure supplied by RUP disciplines. Therefore, our first task consisted in adjusting these disciplines to an SPL context. Then we assigned FSPLP practice areas to these contextualized disciplines in order to achieve further knowledge in each area. Finally, using a technology approach inspired on Software Factories, we used the integrated information to conceive the discipline workflows, artifacts and roles.

| Discipline | Main goal(s) |
|----------------------------|---|
| Domain analysis | Achieve the appropriate knowledge to guarantee a proper set of systems definition (product line) |
| Requirements | Scope refinement through a detailed description of all the (mandatory and optional) features presented in the product line systems |
| Analysis and Design | Three different types of outputs are expected in this discipline: a common architecture definition, the description of the components that assure the pre-established variability and the production process that stipulates the route to assemble the components into the common architecture. |
| Implementation | Implementation of the elements proposed by the solution architect, like the common architecture, the components, tools and extensions to existing tools, DSMLs or generators. |
| Test | The implemented elements have to be tested independently and collectively to assure that the desired functionality is operating as expected and that the correspondent requirement is addressed properly. |
| Deployment | Appropriate delivering of the production infrastructure for the application engineering team. All the components and tools have to be available to the application team that has to hold all the necessary knowledge to operate them. |

Table 1 - SPLUP Domain Engineering core disciplines

Table 1 summarizes the main goal associated with each of the six disciplines that are directly related to the development of the production platform and correspondent variable components. Additionally, the SPLUP Domain Engineering process considers three others disciplines more

focused on the product line program management. The main purpose of this disciplines are presented on Table 2.

| Discipline | Main goal(s) |
|--|--|
| Project Management | Includes the overall program management, including: <ul style="list-style-type: none"> – Product line scoping; – Verification of top management objectives alignment; – Planning, risk identification and mitigation; – Iterations/phases work management. |
| Configuration and Change Management | Architectures and components versioning management (domain engineering point of view) and applications produced versioning management (application engineering point of view). |
| Environment | Provide the software development organization with the software development environment -- both processes and tools -- that will support the development team. |

Table 2 - SPLUP Domain Engineering management disciplines

5.2. SPLUP Application Engineering

The application engineering process is responsible for the application production based on the means delivered by the Domain Engineering team.

According to SLUPUP, any application project must be developed in a short period of time, with “micro” iterations, because (almost) all components that will be included in the solution are already developed. Moreover, unlike the domain process, in the applications projects the customer has a massive participation and must be the center of attention. These were two of the reasons that guided us to adopt a more agile approach to this process rather than the formal RUP approach that shaped the domain process. Due to the RUP similarities and inspirations we chose OpenUp to support the process definition. Unlike SPLUP Domain Engineering/RUP and due to the high application process specificity, the SPLUP Application Engineering has considerable changes in the disciplines structure when compared to the original OpenUp. Only 4 disciplines were set (as described above) and a discipline intending project management was not included because was considered in the domain process. Just like we did in domain process, the next step was assigning FSPLP practice areas to the disciplines. Then, with the information properly integrated we determined the disciplines workflow, artifacts and roles. Table 3 summarizes the application process disciplines.

| Discipline | Main goal(s) |
|---------------------------|--|
| Problem Analysis | Performed by a business analyst that is responsible for: <ul style="list-style-type: none"> – Finding the best fit of the customer needs with the existing product line (this includes negotiating features with the customer and trigger the change request process). – Documenting the customer feedback about the SPL |
| Additional Implementation | This discipline is only applicable in projects that require new (project specific) developments. In these situations, this discipline represents a short stream of development that must be planned and executed to analyze, implement and test the new functionality/component. |

| | |
|------------------------|--|
| Production and tests | Execute the production plan to assemble the common architecture and components, in order to achieve the functionality agreed with the customer through the problem analysis discipline. |
| Application Deployment | This discipline focuses the delivering of the produced application to the customer, including the application installation procedures but also the documentation and any training initiatives. |

Table 3 – SPLUP Application Engineering disciplines

6. SPLUP Application

To demonstrate the SPLUP approach a project was developed. We chose to formulate a case study involving a software house that decided to move from the traditional method to a product line development using SPLUP.

The selected product line domain was student information management in an academic institution. SPLUP guided the entire process, including the domain analysis, the design and implementation of the product line factory⁴ and the resulting application production and deployment. The detailed presentation of this study is available in the MSc Dissertation that sustains this research.

In conclusion, our work on SPLUP showed us that the SPL paradigm, frameworks, model-driven development and generators are entirely compatible.

7. Conclusions

We start our conclusions by recalling the paper's title *Software Product Lines: a realistic path to software development industrialization?*

The process maturity that some SPL oriented organizations exhibit, force us to respond affirmatively. In such organizations, massive reuse is a reality and the development procedure is executed like in any other industry assembly line. The embedded software is a good example of a software industry segment where this achievement can be extensively verified.

But, what about the traditional developers; are SPL a realistic approach to evolve their practices?

Once more, there are organizations that successively adopted a product line approach. Therefore, the immediate answer could be “yes, SPL are a realistic alternative approach to traditional software development”. In Portugal, companies like Primavera⁵ or Quidgest⁶ have been conducting successful product line developments for enterprise management over the last decade.

Nevertheless, even considering the mentioned successful cases, in our opinion the current scenario does not correspond to a real industrialized development. Our perspective is that this “pure” industrialization will only turn reality when at least:

- A software component industry emerges
In the software development, we tend to develop almost every component we need in our application. This is a fairly considerable investment. When we develop reusable

⁴ A demonstration of the prototype is available at <http://www.youtube.com/watch?v=zQg7HmstERE>

⁵ <http://www.primaverabss.com/>

⁶ <http://www.quidgest.pt/>

components, the investment is divided by the future reutilizations but the counterpart is the additional effort associated with reusable components. In most of the industries we know (e.g. hardware, food or automobile industry) the products result from intra-organizational development but also include a big part of third parties components. In today's software development we use platforms - like Microsoft .Net framework or Oracle J2EE - and tools - like Integrated Development Environments (IDEs) or compilers - to create our application but all the remaining assets are developed in-house. Thus, our vision of industrialized development encompasses domain specific component providers to allow creating applications with less direct in-house effort;

- Standards became real standards

Standards are an important matter in all industries because they represent a platform of understanding. This is crucial to establish partnerships that allow to alleviate the internal effort. Over the last decade we witnessed a true exponential growth of organizations exclusively dedicated to software standards definition. What appeared to be a good thing rapidly transformed the lack of standards in a jungle of "fake" standards. Steps in the direction of regulation are mandatory to ensure a proper understanding environment.

Meantime, while these transformations do not occur, reuse centric approaches like SPL, and promising techniques like model-driven development or automation mechanisms can, and should be applied to improve current software development methods towards a "semi-industrialized" process.

8. References

- Brown, A. W., Conallen, J., & Tropeano, D. (2005). Introduction : Models , Modeling , and Model-Driven Architecture (MDA). *Model-Driven Software Development*.
- Clements, P., & Northrop, L. (2001). *Software Product Lines: Practices and Patterns* (3rd ed., p. 608). Addison-Wesley Professional.
- Czarnecki, K., & Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications* (1st ed., p. 864). Addison-Wesley Professional.
- Demir, a. (2006). Comparison of Model-Driven Architecture and Software Factories in the Context of Model-Driven Development. *Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES'06)*, 75-83. Ieee. doi: 10.1109/MBD-MOMPES.2006.5.
- Eveleens, L., & Verhoef, C. (2010). The Rise and Fall of the Chaos Report. *Project Management Focus*.
- Fayad, M. (1997). Object-Oriented Application Frameworks. *ACM Computing Surveys*, 32(1es), 28-es. doi: 10.1145/351936.351964.
- Greenfield, J., Short, K., Cook, S., & Kent, S. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools* (1st ed., p. 665). Wiley Publishing Inc.

- Harsu, M. (2002). A survey on domain engineering. *Research Report (Tampere University Technology)*, 27.
- Johnson, R. E. (1991). Designing Reuseable Classes. *Information Systems*, (217), 1-27.
- Kelly, S., & Tolvanen, J.-P. (2007). *Domain-Specific Modeling* (1st ed.). Wiley Publishing Inc.
- Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture(TM): Practice and Promise* (p. 192). Addison-Wesley Professional. Retrieved November 5, 2010, from <http://www.amazon.com/MDA-Explained-Architecture-Practice-Promise/dp/032119442X>.
- Lenz, G., & Wienands, C. (2006). *Practical Software Factories in .NET* (p. 213). Apress.
- McIlroy, M. D. (1968). Mass Produced Software Components, *Software E*(October 1968), 1-12.
- Mellor, S. J., Scott, K., Uhl, A., & Weise, D. (2004). *MDA Distilled* (p. 176). Addison-Wesley Professional. Retrieved November 5, 2010, from <http://www.amazon.com/MDA-Distilled-Stephen-J-MELLOR/dp/0201788918>.
- Pfleeger, S. L., & Atlee, J. (2006). *Software Engineering - Theory and Practice*. (P. P. Hall, Ed.) (p. 715).
- Pohl, K., Böckle, G., & Linden, F. van der. (2005). *Software product line engineering: foundations, principles, and techniques* (p. 467). Springer.
- Rise, T., & Figures, C. R. (1994). project management The Rise and Fall of the Chaos Report Figures. *Quality*.
- Santos, A. L., Koskimies, K., & Lopes, A. (2008). Automated Domain-Specific Modeling Languages for Generating Framework-Based Applications. *2008 12th International Software Product Line Conference*, 149-158. Ieee. doi: 10.1109/SPLC.2008.17.
- Software Engineering Institute, C. M. (2010). Framework for Software Product Line Practice. Retrieved March 25, 2011, from <http://www.sei.cmu.edu/productlines/>.
- Sommerville, I. (2006). *Software Engineering* (8th ed., p. 867). Addison Wesley.